Software Framework for Data Fault Injection to Test Machine Learning Systems

Nurminen, Jukka K

IEEE
2019-10-28

Nurminen , J K , Halvari , T , Harviainen , J A M , Mylläri , J , Röyskö , A J , Silvennoinen , J & Mikkonen , T 2019 , Software Framework for Data Fault Injection to Test Machine Learning Systems . in Proceedings of 2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE 2019) Workshops : 4th IEEE International Workshop on Reliability and Security Data Analysis (RSDA 2019) . IEEE , pp. 294-299 , International Symposium on Software Reliability Engineering (ISSRE 2019) - 4th IEEE International Workshop on Reliability and Security Data Analysis (RSDA 2019) , Berlin , Berlin , Germany , 28/10/2019 . https://doi.org/10.1109/ISSREW.2019.00087

http://hdl.handle.net/10138/312353
https://doi.org/10.1109/ISSREW.2019.00087

acceptedVersion

Software Framework for Data Fault Injection to Test Machine Learning Systems

Jukka K. Nurminen, Tuomas Halvari, Juha Harviainen, Juha Mylläri,
Antti Röyskö, Juuso Silvennoinen, and Tommi Mikkonen
Department of Computer Science, University of Helsinki, Finland
Email: first.[initial.]last@helsinki.fi

***Abstract*—Data-intensive systems are sensitive to the quality of data. Data often has problems due to faulty sensors or network problems, for instance. In this work, we develop a software framework to emulate faults in data and use it to study how machine learning (ML) systems work when the data has problems. We aim for flexibility: users can use predefined or their own dedicated fault models. Likewise, different kind of data (e.g. text, time series, video) can be used and the system under test can vary from a single ML model to a complicated software system. Our goal is to show how data faults can be emulated and how that can be used in the study and development of ML solutions.**

I. Introduction

Data-intensive systems are sensitive to the quality of data. In deployed systems, data frequently has problems: sensor readings are drifting, network connection problems create gaps in IoT data, textual data is corrupted because of OCR errors, and so on. To ensure that systems are working correctly in different conditions, we need to improve our understanding of how different problems in input data (data faults in the terminology of this paper) influence predictions and other results of systems. Especially in machine learning (ML) systems, we face questions that not only influence the testing phase but also the development decisions. Such questions include the following:

- Should we train the system with perfect or with faulty data? Examples of faulty data are far less common than examples of correct data but we may still have a good understanding of the kinds of data faults the system will face over its lifetime.
- Are some ML algorithms, architectures, or hyperparameter selections more robust towards data faults than others?
- How trustworthy the results of the algorithms are when used in real-life settings, which include faulty input data?

The difficulty of making a system deal with data faults comes from multiple sources. To begin with, data faults come in different forms. Some of them are systematic (e.g. sensor drift), whereas others are more random (e.g. a broken network connection). They happen infrequently so the training material there may not have many examples of faulty cases. Furthermore, it is not obvious what we should do to deal with faults – change the associated training data, change the model, or simply ignore the faulty output somehow.

Unfortunately, testing how a system behaves with different kinds of data faults has been difficult. Especially the rare cases of multiple simultaneous faults are hard to test with real measured data. To solve related problems, the importance of dealing with faulty data has been recently observed. E.g. Qu et al. [1] investigate how faulty data influences the operation of a set of classification, clustering, and regression algorithms. Some tools e.g. [2] have been developed to find and fix problems in datasets. While these tools allow fixing problems in datasets there does not seem to be tools that allow injection of faults to the datasets. Adversarial ML [3] investigates how an attacker with cleverly modified data can cause faulty behavior of ML models. The main idea of our implementation is not to consider an explicitly malicious adversary but to generate datasets that encompass typical faults in data. Similar ideas for hardware and software fault injection have a long history in the development of dependable systems [4].

The approach we are suggesting is to use an artificial fault injector, which modifies the system input data according to some rules. These rules, which form the fault model, should characterize the typical problem situations the system is likely to encounter. We can then use the modified data to test the operation of the system in the faulty conditions, or we can use the faulty data in training to aim for more robust systems.

In this paper, we propose an approach to artificially inject faults to input data to test and to improve the training of ML systems. As a technical contribution, we describe our implementation of a prototype system dpEmu for the task. DpEmu aims to be a flexible and extendable software framework, which allows users to parameterize predefined fault models or introduce their own, inject faults to different kinds of datasets, and study the behavior of single ML models or complicated ML systems. Finally, we show an example of what kind of observations the system allows.

II. Background and Related work

As observed by Breck et al. [5] "Software testing is well studied, as is machine learning, but their intersection has been less well explored in the literature". Recently, however, the situation has started to change. As evidenced by recent surveys [6], [7] researchers have realized the importance of testing in the development of artificial intelligence (AI), including in particular machine learning (ML) systems. The key observation is that ML systems cannot be tested in the same way as classical software systems. Instead, new approaches are needed.

Focusing on data quality for testing is one such approach. Qu et al. [1] suggest a set of guidelines for algorithm selection and data cleaning based on their evaluation of classification, clustering, and regression algorithms. They also observe that unnecessary cleaning of dirty data can be wasteful and therefore understanding what is appropriate cleaning and appropriate algorithm in different cases is important. To enable this they encourage the development of models of how fault types, fault rates, data size, and algorithm influence the performance. Our framework is one way to allow such models to be created and analyzed.

Several tools, e.g. [2], have been developed to find and fix problems in datasets. Li et al. [8] have developed a benchmark, CleanML, to investigate the effect of data cleaning and algorithm selection. They observe than data cleaning alone does not improve performance and can even be harmful. Instead selection of the proper model is important for cleaning approaches to effective. These observations seem to highlight the importance of experimenting with alternative algorithms and data cleaning approaches in the development of ML systems. An explicit goal of the dpEmu system is to make such exploration easier.

Many papers investigate fault injection. They are typically focused on embedded systems and looking at ways for software to allow recovery of hardware problems (see e.g. [9]). Our work is related, but instead of system hardware causing the problem, our faults arise from problems in external data collection and manipulation systems and are reflected as problems in the data that the system is using.

A system close to our thinking is AVFI [10], which is used for injecting faults to systems controlling autonomous vehicles. Its focus is on system-level analysis of autonomous vehicle behavior and, similarly to our work, it allows studying how different sensor faults influence the operation.

DeepRoad [11] uses GAN to artificially generate various weather conditions to road scenes. It is intended for testing autonomous driving systems. It found thousands of problems in state-of-the-art self-driving systems highlighting the importance of thorough testing of these systems. DeepRoad studied how weather and natural conditions influence object detection software while our prime target has been modifications to data because of the technical problems or inadequacies.

DeepMutation [12] is another system close to our thinking. In DeepMutation the system generates artificial faults to both data and model implementation. Many of the mutation operators they suggest are similar to ours.

III. Towards Data Problems Emulation: dpEmu

ML systems are typically developed with pipelines. Data acquisition is followed by data cleaning, preprocessing, training the model, and evaluating the result. The steps can vary and be more detailed depending on the case. To establish the data fault injection as part of the development process we feel that tools, which merge data fault testing and analysis into the development pipeline, are essential. This observation was the main motivation for our work towards the development of a software framework, illustrated conceptually in Fig. 1, with the following goals:

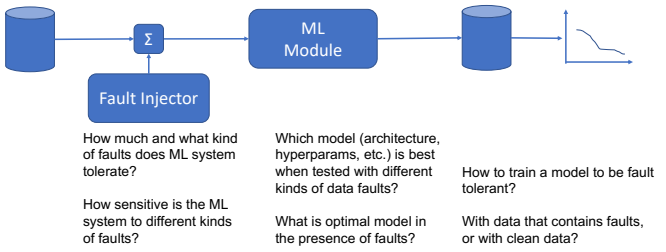

Fig. 1: Conceptual view of the system with examples of questions that the system can answer.

- Easy and flexible modeling of the types of data faults the system is likely to encounter via a combination of predefined parameterizable fault models and new user-defined ones. Ideally, the set of predefined fault models should gradually grow as a result of the integration of new fault models for new purposes.
- Ability to work with different kinds of structured and unstructured data as well as with highly different ML models or systems.
- Parameterization of the data fault generation so that developers can study how sensitive their systems are to different kinds of faults and what are the thresholds of data problems when the system starts to lose its performance.
- Visualization of the results with different fault models and parameters.
- Bookkeeping to allow going back to the sources of problems.
- Embedding the fault injection and the visualization of the effects of faulty data to the development pipeline.
- Integration of data fault emulation to different development pipelines.

To satisfy these goals, we have created a generator framework for emulating data problems, called dpEmu. The framework can generate faults in training or testing data in a controlled and documentable manner, and it enables emulating data problems in the use and training of ML systems as depicted in Fig. 2. The Runner routine introduces faults in a dataset, following the definitions set in the Fault generation tree. Then, the resulting data is preprocessed and run by ML models, which produces final results that can be visualized to the user.

DpEmu can run one or more ML models on any data using different values for the fault parameters and visualize the results. Written in Python, dpEmu can easily interact with any Python-based ML framework such as Sklearn, TensorFlow, or PyTorch. It is available as open-source at https://github.com/dpEmu/dpEmu.

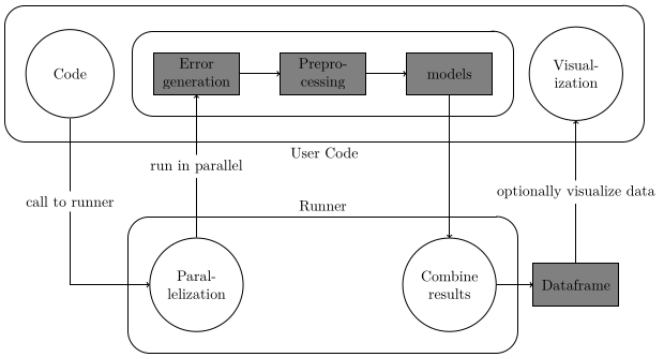

Fig. 2: DpEmu pipeline.

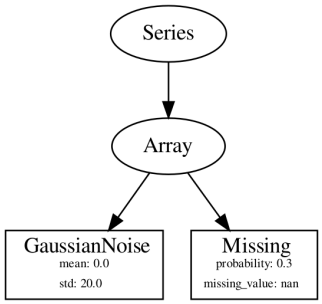

Fig. 3: Visualization of a possible fault generation tree with added Gaussian noise and missing pixels.

(a) Original (b) With added noise (c) With added noise and missing pixels

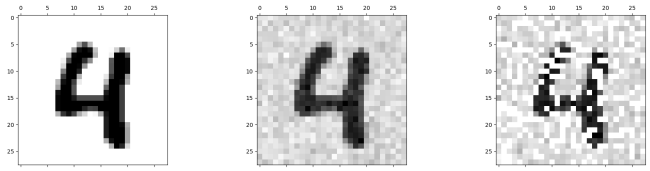

Fig. 4: Sample MNIST figure processed with filters.

A. *Fault Generation Tree*

Fault injection in dpEmu is based on fault generation trees. The aim is a representation which is easily expandable and can be applied to datasets of almost any kind. First we construct a tree characterizing the structure of the data. Then we apply a set of transformations to specific parts of the data by adding Filters to selected leaf nodes. The Filters specify the transformations we want to make to the data. It is possible to use predefined Filters but users can also program their own Filters to extend the framework and create new transformations that inject faults specific to their problems.

Data is usually characterized by a combination of Array, Tuple, Series, and TupleSeries nodes. A Series node often forms the root of the tree. An Array node can represent a NumPy array or a Python list of any dimension. It can even represent a scalar value provided it is not the root of the tree. By combining different nodes we can create more complex representations. For example, we may choose to represent a matrix as a series of row arrays: `root_node = Series(Array())`.

Data points stored as tuples (such as those characterizing .wav audio) can be represented using a Tuple node. A TupleSeries represents a tuple of data items whose k-th elements are tightly related. For example, if we have NumPy array X containing the inputs and array Y containing the corresponding labels, we may choose to represent (X, Y) as a TupleSeries.

There is typically more than one valid way to represent the structure of the data as a tree. For example, a two-dimensional NumPy array can be represented as a matrix, i.e. an Array node; as a list of rows, i.e. a Series with an Array as its child; or as a list of lists of scalars, i.e. a Series whose child is a Series whose child is an Array.

Filters, which act as the fault sources, can be added to leaf nodes. Filters can have parameters, which typically control the severity of the injected faults. Dozens of Filters, such as Snow, Blur, and SensorDrift, are predefined. They can be used to manipulate data of various kinds, including images, time series, and sound. Users can also create their own custom fault sources by subclassing the `Filter` class.

As an example, let us consider the MNIST dataset (https://www.openml.org/d/554) of handwritten digits. The input consists of 70000 rows where each row is a 784 pixel (i.e. 28×28) black and white image of a handwritten digit (Fig. 4a). The first step is to form the fault generation tree. Since the input is an indexed collection of images, we can represent it as a series of arrays, with each array corresponding to a single image. We can then add Filters to manipulate the images, as shown below.

```
image_node = Array()
series_node = Series(image_node)
image_node.addfilter(GaussianNoise("mean", "
    std"))
```

The GaussianNoise filter adds noise drawn from a normal distribution. The constructor takes two arguments, which are identifiers for the parameters. We can now provide values for these parameters and generate the faulty data:

```
params = {"mean": 0.0, "std": 20.0}
err_data = series_node.generate_error(data,
    params)
```

The resulting image with added noise is shown in Fig. 4b.

We are not limited to one fault type per node. Below, we add another filter, Missing, which changes each array element to a user-specified value such as NaN ("not a number") with the provided probability:

```
image_node.addfilter(Missing("probability", "
    missing_value"))
params = {"mean": 0.0, "std": 20.0, "
    probability": .3, "missing_value": np.nan}
err_data = series_node.generate_error(data,
    params)
```

The resulting fault generation tree is shown in Fig. 3. The modified image with injected noise and missing pixels is presented in Fig. 4c.

B. Exploratory Execution

To support exploratory use, dpEmu includes a Runner system. It creates subprocesses for each set of fault parameters, and in each subprocess, all of the included ML models are run, with possibly multiple sets of hyperparameters. This allows for distributing the processing. When all subprocesses are finished running, the system returns the results as a pandas DataFrame object which can then be used for visualization.

The runner takes the following inputs when it is run: training data, test data, a preprocessor class, preprocessor parameters, a fault generation root node, a list of fault parameters and a list of ML models and their parameters:

```
df = runner.run(
    train_data=train_data,
    test_data=test_data,
    preproc=Preprocessor,
    preproc_params={},
    err_root_node=get_err_root_node(),
    err_params_list=get_err_params_list(),
    model_params_dict_list=
        get_model_params_dict_list()
)
```

The list of fault parameters is simply a list of dictionaries, which contain the keys and fault values for the fault generation tree. The list of ML model parameters is a list of dictionaries containing three keys: "model", "params_list" and "use_clean_train_data". The value of "model" is the name of a class where the actual model is called. The value of "params_list" is a list of dictionaries where each dictionary contains one set of parameters for the model. The model is run with all of these parameter combinations in each subprocess. If "use_clean_train_data" is true, Runner will always pass the original, clean training data to the model in addition to the test data with injected faults. If there is no training data, a None value can also be passed to the Runner.

The user-defined preprocessor is run twice in every subprocess, right after the fault generation, using both clean and faulty training data so that the correct version of the training data can be passed to each model. The preprocessor implements a function run(train_data, test_data, params), and it returns the preprocessed train and test data. The preprocessor can return additional data as well, and it will be listed as separate columns in the DataFrame which the runner returns:

```
class Preprocessor:
    def run(self, train_data, test_data,
            params):
        return train_data, test_data, {}
```

Each model class should implement function run(train_data, test_data, parameters) which optionally trains the model on the training data and tests the model with test data with given model parameters and returns a dictionary containing the scores and possibly additional data to be added to the resulting DataFrame:

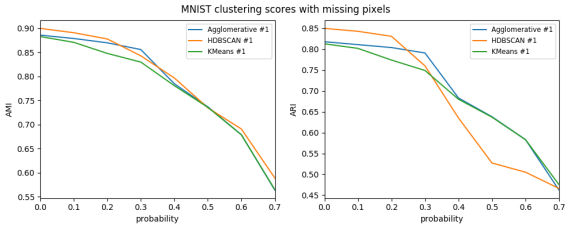

Fig. 5: Visualization of AMI and ARI scores for three models when clustering MNIST dataset with missing pixels at different fault levels. The scores for HDBSCAN are hyperparameter-optimized.

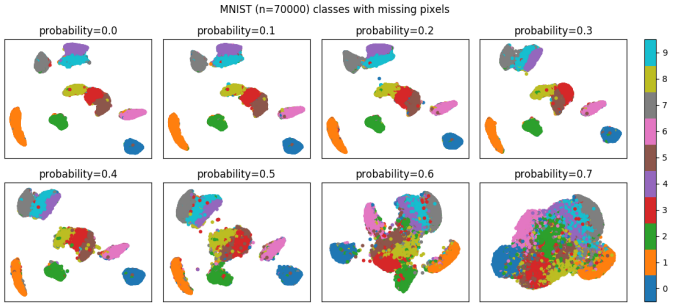

Fig. 6: 2D visualization of the MNIST dataset with different fault probability levels.

```
class Model:
    def run(self, train_data, test_data,
            params):
        return {}
```

All of the data given to the specific model has first passed through fault generation and preprocessing. You can, for example, use the preprocessor to write the faulty data to files and then call the CLI of an ML model using our run_ml_module_using_cli(cline), which returns the output of an external model to be parsed.

C. Visualization

In addition to fault generation, dpEmu can be used for visualizing the results of the desired parameter combinations. It supports visualizing (i) hyperparameter-optimized scores for each model at different fault levels (Fig. 5); (ii) interactive plots where data points can be clicked to visualize it; (iii) 2D visualization of the dataset at different fault levels using original labels (Fig. 6); (iv) the model parameter values which give the best results; (v) interactive confusion matrices of the classification results (Fig. 7); and (vi) the fault generation tree.

D. Use Cases

To test the usefulness of dpEmu we have studied several use cases with different kinds of data and algorithms. These studies include the following:

– Comparing some clustering algorithms' performance when clustering images from datasets like MNIST or Fashion MNIST while adding different amounts of faults,

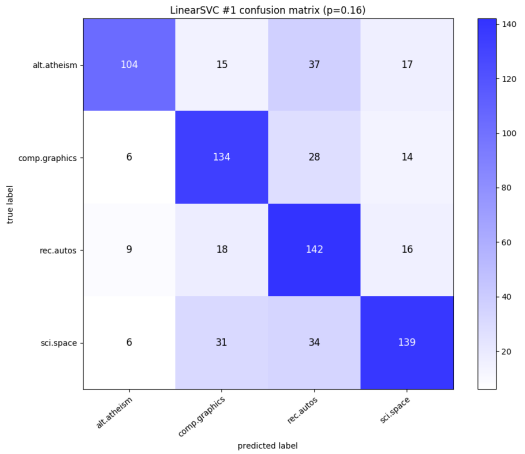

Fig. 7: Visualization of a confusion matrix for a LinearSVC model from a 20 Newsgroups dataset with areas.

for example Gaussian noise or missing pixels, to the images. Dimensionality reduction is applied to the images in the preprocessing phase before clustering. The evaluation is done by calculating AMI and ARI scores using the original labels provided. Also in the case of algorithms like HDBSCAN, optimal hyperparameters at varied fault levels are studied.

– Comparing different classification algorithms' performance when classifying texts from the 20 newsgroups dataset while training the models with both clean and faulty data at different fault levels. Fault sources such as OCR faults and random missing areas are used. Optimal hyperparameters at different fault levels are also studied.
– Forecast future values of time series using the LSTM model when measured values have (i) arbitrary faults; (ii) systematic drift, and (iii) gaps.
– Accuracy of the recognition of spoken commands as a function of the dynamic range of the audio.
– Object detection, which we will discuss in the next section in detail.

Results for different use cases are available at https://dpemu.readthedocs.io/en/latest/index.html#case-studies.

IV. Object Detection Case Study

As a concrete example, we next use dpEmu to analyze the accuracy of different object detection models. The code and more details of the analysis are available at https://dpemu.readthedocs.io/en/latest/index.html#case-studies. In the study, we compared the performance of three models from FaceBook's Detectron project (FasterRCNN, MaskRCNN, and RetinaNet) and YOLOv3 model from Joseph Redmon, when different fault sources were added. We used 118 287 jpg images (COCO train2017) as training data and 5000 jpg images (COCO val2017 http://cocodataset.org/#download) as test data to calculate the mAP-50 scores. We used the pre-trained weights for FasterRCNN (e2e_faster_rcnn_X-101-64x4d-FPN_1x), MaskRCNN (e2e_mask_rcnn_X-101-64x4d-FPN_1x) and RetinaNet (retinanet_X-101-64x4d-FPN_1x) from Detectron's model zoo. YOLOv3's weights were trained by us, using the Kale cluster of the University of Helsinki. The training took approximately five days when two NVIDIA Tesla V100 GPUs were used.

(a) Example of added rain. (b) Example of added snow.

Fig. 8: Examples of rain and snow filters.

(a) Gaussian blur (b) Rain

(c) Snow (d) Resolution

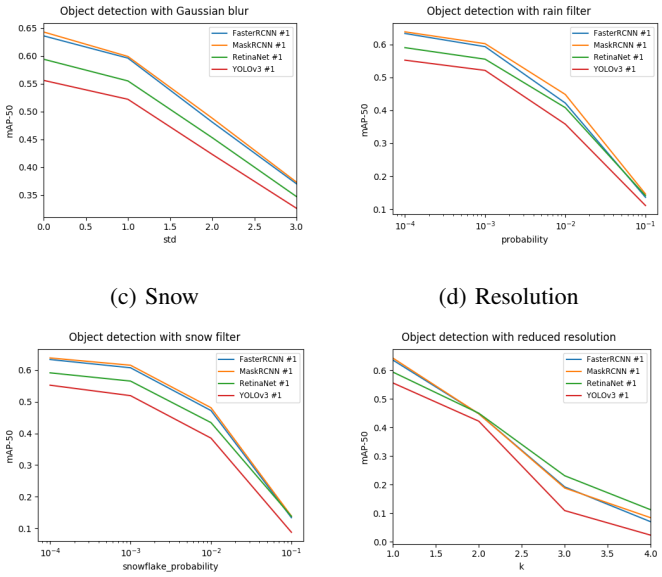

Fig. 9: Effect of different filters to the accuracy of object detection algorithms.

For dpEmu, we have defined several Filters to generate different kinds of faults in images, including Gaussian blur, Added rain, Added snow, and Resolution change. The Gaussian blur filter added normally distributed noise with mean 0 to the data. The standard deviation parameter was varied. Added rain and snow filters introduced simulated rain and snow to images. Sample effects of the Added rain and snow filters can be seen in Fig. 8. The resolution was changed with the formula:

$$\text{image}[y][x] = \text{image}\left[k\left\lfloor\frac{y}{k}\right\rfloor\right]\left[k\left\lfloor\frac{x}{k}\right\rfloor\right]$$

The results of the object detection are presented in Fig. 9. As expected, the figure shows that object detection accuracy

drops when the amount of disturbance in the picture increases. Because it is difficult to compare the severity of different kinds of faults (the x-axes), it is difficult to state which type of fault is the most harmful. However, we can see that with Gaussian blur, Added rain, and Added snow (Fig. 9a, 9b, 9c), the order of the different models remains the same, and MaskRCNN is the most accurate with all three fault types. With Added rain, FasterRCNN accuracy drops faster than the accuracy of MaskRCNN.

Interestingly, with reduced resolution (Fig 9d), the accuracy order changes as the resolution gets worse. RetinaNet, which initially was 3rd, gave more accurate results than FasterRCNN and MaskRCNN, which initially were more accurate. These results are tentative and a more extensive analysis would be useful to evaluate the merits of different models in changing conditions. They are, however, demonstrators of the kinds of analysis dpEmu supports.

V. Conclusions

To develop robust and reliable ML systems we have created dpEmu to encourage developers to evaluate how their models and systems work when system input data has faults. The system can be used for multiple purposes, such as investigating how a trained model or an entire system tolerate different kinds of faults in its input data; studying which model and hyperparameterization are the best when input data has certain kinds of faults or how alternative data cleaning approaches influence the operation of the resulting model; evaluating trade-offs between model accuracy versus model robustness; and quantifying the accuracy difference when the model is trained with clean or faulty data. At present, dpEmu still is a prototype and as usual, it is not perfect in terms of functionality and usability. However, it already acts as demonstrator regarding how robustness and tolerance to data faults can be integrated into the development pipeline of ML systems.

Popular ML libraries, such as Sklearn or TensorFlow, have extensive collections of functions for evaluating the models as well as splitting the datasets for training and testing parts. However, none of these, seem to have built-in support for studying the model behavior in case of erroneous input data. DpEmu can be used together with these libraries to add one step before the actual training of the model. The addition of one more step, however, will increase the training effort a lot. In addition to the actual training, it is possible to have another training loop that searches for the best possible architecture and hyperparameterization for the system [13]. Introduction of a third loop with different data faults to ensure the system works in optimal, or adequate, fashion also with data faults will further increase the already massive computational effort.

To ensure that ML models, and systems based on those, are robust it is important to test how the systems work when input data has faults. We envision that in the future models of typical faults would exist for all regularly used data, such as for the behavior of different kinds of sensors and their connectivity. Such models already exist e.g. for optical character recognition (OCR) [14]. The developers would then choose among different predefined fault models and parameterize them to match their needs. If a ready-made fault model is missing, users could create their own. For their maintenance, ML systems could not only track their behavior as they do now but also track the behavior of the fault model that was used in their development. A deviation from the fault model behavior could be an indication of the need to reconsider if the model in use is still optimal for the newly observed behavior.

References

[1] Z. Qi, H. Wang, J. Li, and H. Gao, "Impacts of Dirty Data: and Experimental Evaluation," 3 2018. [Online]. Available: http://arxiv.org/abs/1803.06071

[2] N. Hynes, D. Sculley, G. Brain, M. T. Google Brain, and M. Terry, "The Data Linter: Lightweight, Automated Sanity Checking for ML Data Sets," in *NIPS MLSys Workshop*, 2017. [Online]. Available: https://github.com/brain-research/data-linter

[3] L. Huang, A. D. Joseph, B. Nelson, B. I. Rubinstein, and J. D. Tygar, "Adversarial machine learning," in *Proceedings of the 4th ACM workshop on Security and artificial intelligence - AISec '11*. New York, New York, USA: ACM Press, 2011, p. 43. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2046684.2046692

[4] H. Ziade, R. Ayoubi, and R. Velazco, "A Survey on Fault Injection Techniques," Tech. Rep. 2, 2004. [Online]. Available: https://citemaster.net/get/f8626be0-10dd-11e6-a12d-00163e009cc7/04-Hissam.pdf

[5] E. Breck, S. Cai, E. Nielsen, M. Salib, and D. Sculley, "What's your ML Test Score? A rubric for ML production systems," in *NIPS Workshop on Reliable Machine Learning in the Wild*, 2016.

[6] H. B. Braiek and F. Khomh, "On Testing Machine Learning Programs," 12 2018. [Online]. Available: http://arxiv.org/abs/1812.02257

[7] J. M. Zhang, M. Harman, L. Ma, and Y. Liu, "Machine Learning Testing: Survey, Landscapes and Horizons," 6 2019. [Online]. Available: http://arxiv.org/abs/1906.10742

[8] P. Li, X. Rao, J. Blase, Y. Zhang, X. Chu, and C. Zhang, "CleanML: A Benchmark for Joint Data Cleaning and Machine Learning [Experiments and Analysis]," 4 2019. [Online]. Available: http://arxiv.org/abs/1904.09483

[9] M. Kooli and G. Di Natale, "A survey on simulation-based fault injection tools for complex systems," in *2014 9th IEEE International Conference on Design & Technology of Integrated Systems in Nanoscale Era (DTIS)*. IEEE, 5 2014, pp. 1–6. [Online]. Available: http://ieeexplore.ieee.org/document/6850649/

[10] S. Jha, S. S. Banerjee, J. Cyriac, Z. T. Kalbarczyk, and R. K. Iyer, "AVFI: Fault Injection for Autonomous Vehicles," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. IEEE, 6 2018, pp. 55–56. [Online]. Available: https://ieeexplore.ieee.org/document/8416212/

[11] M. Zhang, Y. Zhang, L. Zhang, C. Liu, and S. Khurshid, "DeepRoad: GAN-based metamorphic testing and input validation framework for autonomous driving systems," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering - ASE 2018*. New York, New York, USA: ACM Press, 2018, pp. 132–142. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3238147.3238187

[12] L. Ma, F. Zhang, J. Sun, M. Xue, B. Li, F. Juefei-Xu, C. Xie, L. Li, Y. Liu, J. Zhao, and Y. Wang, "DeepMutation: Mutation Testing of Deep Learning Systems," in *Proceedings - International Symposium on Software Reliability Engineering, ISSRE*, vol. 2018-October. IEEE Computer Society, 11 2018, pp. 100–111.

[13] R. Miikkulainen, J. Liang, E. Meyerson, A. Rawal, D. Fink, O. Francon, B. Raju, H. Shahrzad, A. Navruzyan, N. Duffy, and B. Hodjat, "Evolving Deep Neural Networks," *Artificial Intelligence in the Age of Neural Networks and Brain Computing*, pp. 293–312, 1 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780128154809000153

[14] H. S. Baird, "The State of the Art of Document Image Degradation Modelling," in *Digital Document Processing*. Springer, London, 2007, pp. 261–279. [Online]. Available: http://link.springer.com/10.1007/978-1-84628-726-8_12